View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital Library of the Czech Technical University in Prague

Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

Algorithms for Combinatorics on Words Related to Palindromic Richness

Matúš Gura

Supervisor: Ing. Štěpán Starosta, Ph.D.

26th June 2015

Acknowledgements

I would like to express my deepest gratitude to my advisor, Ing. Štěpán Starosta, Ph.D., for his guidance, immense patience, and knowledge.

My sincere thanks also goes to my family and my friends, for their continuous support, and for always being there cheering me up.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 26th June 2015 .....

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Matúš Gura. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Gura, Matúš. *Algorithms for Combinatorics on Words Related to Palindromic Richness*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Pre danú konečnú grupu G skladajúcu sa z morfizmov a antimorfizmov, popisujeme konečné slová s jazykom uzavretým na grupe G. Definujeme súvisiace pojmy a definície, pričom sa zameriavame na G-bohatosť a k tomu úzko súvisiaci pojem G-palindromický defekt. Analyzujeme použitelnosť vybraných algoritmov a dátových štruktúr na implementáciu nenaivného algoritmu na výpočet G-defektu. Implementujeme a testujeme náš návrh, pričom výsledky vykonanıych testov potvrdzujú správnosť nášho algoritmu. Veríme, že v blízkej budúcnosti sa náš algoritmus stane súčasťou softvéru SageMath.

Klíčová slova slová, kombinatorika, palindromická bohatosť, palindromický defekt

Abstract

For a given finite group G consisting of morphisms and antimorphisms, we study finite words with language closed under the group G. We introduce related notions and definitions focusing on G-richness and its related notion of G-palindromic defect. We discuss and consider several algorithms and data structures for the implementation of a non-naive algorithm for G-defect computation. We implement and test our the proposed solution. Tests confirm

the correctness of our solution. We believe that our implemented algorithm will become a part of the SageMath in the near future.

Keywords words, combinatorics, palindromic richness, palindromic defect

Contents

List of Figures

List of Tables

Introduction

Words are the basis of human interaction and the smallest element of language with a literal or practical meaning. It is not a surprise that the notion of word can be found in many ancient mathematical works [3]. Over time, words have become a research topic of their own.

A generally considered starting point of mathematical research on words is the paper on repetition-free words by A. Thue published in 1906 [21]. The systematic research on words culminated in the second half of the 20th century when a group of mathematicians under the pseudonym M. Lothaire released the summarization of the research done until then [14].

In recent years there has been a growing interest in the research of words that read the same forward and backward - palindromes. This interest comes from many areas – from number theory [2] and theoretical physics [11] to genetics [12]. It was observed [8] that a finite word w of length $|w|$ contains at most $|w| + 1$ different palindromes. If the word w contains the maximal number of different palindromes, it is called rich [8, 10]. The difference between the maximal number of distinct palindromes and the actual number of palindromes in w is called the palindromic defect [6].

Attempts to generalize the notion of palindrome and palindromic richness appeared soon [12, 17]. Instead of classical palindromes defined as words invariant under the reversal mapping appeared Θ-palindromes defined as words invariant under an involutive antimorphism Θ [12]. A further generalization was suggested in [16], under the name of G-palindromes. G-palindromes are words invariant under more symmetries [17]. This thesis aims to describe related notions of G-palindromes and to design an algorithm for computing G-palindromic defect.

Main definitions and relevant notions of words, palindromes and generalized palindromes are introduced in Chapter 1. Several interesting string algorithms and data structures are described in Chapter 2. In Chapter 3 a mathematical open source software SageMath and its internal methods relevant to the notion of word are introduced. Chapter 4 investigates possible

implementations of G-palindromic defect. Chapter 5 describes the implemented solution, discusses the final time complexity and presents the results of performance testing.

Chapter 1

Notation and Terminology

1.1 Words

Let $\mathcal{A}$ denote a finite nonempty set of symbols, called the *alphabet*. The elements of $\mathcal{A}$ are *letters* and a (finite or infinite) string formed by letters of $\mathcal{A}$ is a *word*. Let $\mathcal{A}^*$ denote the set of finite words over the alphabet $\mathcal{A}$, and by $\mathcal{A}^+$ the set of all nonempty finite words over the alphabet $\mathcal{A}$:

$$\mathcal{A}^+ = \mathcal{A}^* \setminus \{\epsilon\},$$

where ϵ denotes the empty sequence, called the *empty word*. Any subset of $\mathcal{A}^*$ is called a *language*.

Let w be a finite word defined as follows:

$$w = a_1 a_2 \cdots a_n, \quad a_i \in \mathcal{A} \text{ and } 1 \leq i \leq n.$$

The *length* of w, denoted by $|w|$, is n. We denote by $|w|_a$ the number of occurrences of the letter a in the word w, and by $\mathcal{A}^n$ the set of all words of length n over $\mathcal{A}$:

$$\mathcal{A}^n = \{u \in \mathcal{A} \mid |u| = n\} = \{a_1 a_2 \cdots a_n \mid a_i \in \mathcal{A}\}.$$

The set of all letters occurring in w at least once is denoted by $alph(w)$. Therefore a letter $a \in \mathcal{A}$ belongs to $alph(w)$ if and only if $|w|_a \geq 1$.

A word can be represented as an array of letters. Throughout the thesis, the zero based array indexing is used, when appropriate:

$$w[i-1] = a_i, \quad a_i \in \mathcal{A} \text{ and } 1 \leq i \leq n.$$

Two words $u = a_1 a_2 \cdots a_n$ and $v = b_1 b_2 \cdots b_m$ are *equal* if and only if they are of the same length, and the letters at corresponding positions are the same:

$$m = n \quad \text{and} \quad a_i = b_i, \text{ for } 1 \leq i \leq n.$$

A *monoid* $\mathcal{M} = (M, \circ)$ is a set closed under a binary operation $\circ$ that is associative and it has an identity element. Note that $\mathcal{A}^*$ is a monoid over $\mathcal{A}$. The binary operation of $\mathcal{A}^*$ is *concatenation*. The concatenation of two words $u = b_1 b_2 \cdots b_p$ and $v = c_1 c_2 \cdots c_q$ is the word w such that:

$$w = uv = b_1 b_2 \cdots c_p c_1 c_2 \cdots c_q, \quad |w| = |u| + |v|.$$

The identity element of $\mathcal{A}^*$ is ϵ; hence, for any word u it is true that $\epsilon u = u\epsilon = u, u \in \mathcal{A}^*$.

A word $x \in \mathcal{A}^*$ is a *factor* of w if there exist words $u, v \in \mathcal{A}^*$ such that

$$w = uxv.$$

An index $i \in \mathbb{N}$ is called *occurrence* of x, such that

$$x = w_i u_{i+1} \cdots u_{i+|x|-1}.$$

If $u = \epsilon$, then x is called a *prefix* of w, and if $v = \epsilon$, then x is called a *suffix* of w. We denote by $\mathcal{L}(w)$ the set of all factors of w, and by $\mathcal{L}_n(w)$ the set of all factors of length n of w.

For example, if $z = caba$, then

$$\begin{aligned}
\mathcal{L}_0(z) &= \{\epsilon\} \\
\mathcal{L}_1(z) &= \{c, a, b\} \\
\mathcal{L}_2(z) &= \{ca, ab, ba\} \\
\mathcal{L}_3(z) &= \{cab, aba\} \\
\mathcal{L}_4(z) &= \{caba\}
\end{aligned}$$

A finite word w is a factor of u if there exists an index i

The *reversal* of w is the word

$$R(w) = R(a_1 a_2 \cdots a_n) = a_n a_{n-1} \cdots a_1,$$

where R is the reversal mapping. A word equal to its reversal ($w = R(w)$) is a *palindrome*.

The set of all palindromes in $\mathcal{L}(w)$ is denoted by $Pal(w)$. It was proved [8] that w contains at most $|w|+1$ distinct palindromes (including ϵ). If w contains the maximum number of different palindromic factors it is called *rich*.

For example, if we take $z = caba$, then

$$\begin{aligned}
\mathcal{L}(z) &= \{\epsilon, c, a, b, ca, ab, ba, cab, aba, caba\} \\
Pal(z) &= \{\epsilon, c, a, b, aba\}.
\end{aligned}$$

The word z is rich because $|Pal(z)| = |z| + 1$.

We denote by $\mathcal{D}(w)$ the *palindromic defect*. It is defined as the difference between the maximal number of palindromes and the actual number of palindromes in w [5]:

$$\mathcal{D}(w) = |w| + 1 - |Pal(w)|.$$

In this notion, finite rich words have defect equal to 0.

1.2 Morphisms and Antimorphisms

Let $\mathcal{A}$ and $\mathcal{B}$ be two finite alphabets. A mapping $\varphi : \mathcal{A}^* \to \mathcal{B}^*$ on $\mathcal{A}^*$ is

- a *morphism* if $\varphi(uv) = \varphi(u)\varphi(v)$, for any $u, v \in \mathcal{A}^*$;
- an *antimorphism* if $\varphi(uv) = \varphi(v)\varphi(u)$, for any $u, v \in \mathcal{A}^*$.

We denote the set of all morphisms and antimorphisms on $\mathcal{A}^*$ by $AM(\mathcal{A}^*)$. We denote the set of all morphisms on $\mathcal{A}^*$ by $M(\mathcal{A}^*)$. Note, that the reversal mapping R is an antimorphism. Any antimorphism is a composition of R and a morphism: $R(M(\mathcal{A}^*))$. Thus,

$$AM(\mathcal{A}^*) = M(\mathcal{A}^*) \cup R(M(\mathcal{A}^*)).$$

Given an antimorphism φ, a word w is a φ-palindrome if $w = \varphi(w)$.

1.3 Words with language closed under a group G

The following definitions were introduced in [16] and [17].

All the following notions are defined for G which stands for a finite subgroup of $AM(\mathcal{A}^*)$. Moreover, to generate non-trivial results, we require G to contain at least one antimorphism.

Let $w, v \in \mathcal{A}^*$. The words w, v are *G-equivalent* if there exists $\mu \in G$ such that

$$w = \mu(v).$$

The *class of equivalence* containing w is denoted by $[w]$ and it is a set defined as

$$[w] = \{\mu(w) \mid \mu \in G\}.$$

A word w is *G-palindrome* if there exists an antimorphism $\Theta \in G$ such that

$$w = \Theta(w), \quad \text{i.e., } w \text{ is a } \Theta\text{-palindrome.}$$

G-occurrence of v in w is an index i such that there exists $v' \in [v]$ having occurrence i in w. If there is exactly one G-occurrence of v in w we say v is *G-unioccurrent* in w.

A suffix $u \in \mathcal{A}^*$ of a word w is called a *G-longest palindromic suffix* of w if u is a G-palindrome and there is no longer G-palindromic suffix in w. It is denoted by *G-lps*(w). Note that ϵ is always a suffix of w and a G-palindrome. Thus, if there is no non-empty G-palindromic suffix that satisfies this condition, then *G-lps*$(w) = \epsilon$.

1.4 G-richness

To define the G-analogy of richness we need to recall its connection to the palindromic defect. Let us recall that the palindromic defect is the difference between the maximal number of palindromes and the actual number of palindromes in a word. Thus, a finite word is called G-rich if and only if G-defect is equal to 0.

As described and proved in [17], there are two equivalent characterizations of *G-defect*:

1. We first define the set of all G-palindromic classes of equivalence in w as follows

$$Pal_G(w) = \{[v] \mid v \text{ is a factor of } w \text{ and a G-palindrome}\}.$$

The G-defect is defined as

$$D_G(w) = |w| + 1 - \#\mathrm{Pal}_G(w) - \gamma_G(w) \tag{1}$$

where

$$\gamma_G(w) = \#\{[a] \mid a \in \mathcal{A} \text{ is factor of } w, a \neq \Theta(a) \text{ for all antimorphisms } \Theta \in G\}.$$

2. Let $w = w_1 \cdots w_n \in A^*$. A number i such that $1 \leq i \leq n$ is called *G-lacuna* in w if w_i and G-lps$(w_1 \cdots w_i)$ are not G-unioccurrent in $w_1 \cdots w_i$. Then

$$D_G(w) = \text{the number of G-lacunas in } w. \tag{2}$$

Let us demonstrate both these definitions on an example.

Take a word $w = abbab$ and two antimorphisms: $\Theta_1 = a \mapsto b, b \mapsto a$ and $\Theta_2 = a \mapsto a, b \mapsto b$. Let us prove that w is G-rich by computing its defect.

The set of all distinct factors of w is:

$$\mathcal{L}(w) = \{\epsilon, a, b, ab, ba, bb, abb, bab, bba, abba, bbab, abbab\}$$

We decide whether factors in $\mathcal{L}(w)$ are G-palindromes or not and we generate their classes of equivalence. The list of G-palindromes and their classes of equivalence are:

$\Theta_1(ab) = ab$	$[ab] = \{ab; ba\}$
$\Theta_1(ba) = ba$	$[ba] = \{ba; ab\}$
$\Theta_2(a) = a$	$[a] = \{b, a\}$
$\Theta_2(b) = b$	$[b] = \{a, b\}$
$\Theta_2(bb) = bb$	$[bb] = \{aa, bb\}$
$\Theta_2(bab) = bab$	$[bab] = \{aba, bab\}$
$\Theta_2(abba) = abba$	$[abba] = \{baab, abba\}$

We can see that $[a] = [b]$ and $[ab] = [ba]$. Therefore, the set of all G-palindromic classes of equivalence of w is:

$$Pal_G(w) = \{[\epsilon], [a], [ab], [bb], [bab], [abba]\}.$$

Since there is no letter in w that is not a G-palindrome we have $\gamma_G(w) = 0$.

The final G-palindromic defect can be then computed as follows:

$$D_G(w) = |w| + 1 - \#\mathrm{Pal}_G(w) - \gamma_G(w) = 5 + 1 - 6 - 0 = 0$$

Word w is G-rich.

Chapter 2

Algorithms

This chapter describes data structures and algorithms that have been studied and considered for our implementation. These algorithms are mostly related to string searching.

2.1 Manacher's algorithm

Manacher's algorithm, introduced in [15], is an algorithm for finding the longest palindromic factor of a word in a linear time. Since G-defect can be computed using G-longest palindromic suffixes of a word, we introduce this algorithm to later analyze whether we can use some of its properties for our implementation or not.

A factor q of length m can be either of odd length or even length. Odd length factor q has a centre at the index $\lfloor m/2 \rfloor$. Even length factor q has a centre between two letters at the indices $m/2-1$ and $m/2$. To better illustrate the algorithm and have one index for a factor of any size, we transform q to a word q' by inserting a special letter `#` that is not in $alph(q)$ between every letter of q:

```
q  =   B   A   N   A   N   A
q' = # B # A # N # A # N # A #
```

The transformed word q' has length $2m + 1$ and its centre is at the index m.

The output of this algorithm is an array where every index represents the longest palindrome centered at the index location.

- We set these variables with the initial values 0:
  - `C`: the index of the centre of the current palindrome
  - `R`: the index of the right side of the current palindrome
  - `i`: the index of the current position

  - `i'`: the mirror index of the current position $(2 \times C - i)$

- We iterate through `q'` from the left to the right incrementing value of `i` with every shift.
- For a particular position `C` we compare its neighbours on left (`i'`) and right (`i`). If they are equal, we move further to the right from `C` and we repeat this step, otherwise the longest palindrome centered at `C` is found and its size is written to `P[C]`. We save the index of palindrome's right edge to `R`.
- Palindromes are symmetric. We can use this property and expect that right side of a palindromic factor is the same as its left side. But this is not true if the difference of the index of palindrome's right edge and the current index is greater or equal to the length of the palindrome centered at the mirrored index (`R-1<=P[i']`).
- If the palindrome centered at `i` expands past `R`, we update `C` to `i` and extend `R` to the new palindrome's right edge.

2.2 Boyer-Moore Algorithm

Boyer-Moore algorithm introduced in [4] is a string searching algorithm. The algorithm has been studied and described in the thesis because it is considered as "the most efficient string matching algorithm in usual applications" [7]. In this section we compare the algorithm to the naive solution and we describe its basic properties and rules.

Given a text t and a word p called *pattern*, over a finite alphabet Σ, the algorithm finds an occurrence of the pattern in t. Set $m = |p|$ and $n = |t|$.

We will consider here a simpler version of this problem, when our goal is just to find the first occurrence of p in t since it is more illustrative and it can be adapted easily for the more general problem.

The naive string search algorithm runs through all positions in t and checks whether an occurrence of p starts there or not. A pseudocode of this approach is shown in Algorithm 2.1. As one can see in Algorithm 2.1, the outer loop runs at most $n - m + 1$ times and the inner loop runs at most m times. Thus, the time complexity of the naive string search algorithm is clearly $O(nm)$ in the worst case (such as $p = aaab$, $t = aaaaaaaab$). With this naive approach we always shift p by one letter to the right when the comparison fails.

In Boyer-Moore algorithm the start of p is aligned with the start of t:

```
t = I G U A N A S   E A T   B A N A N A S
p = B A N A N A
```

Words are matched from the end of p to the start of p. If there is a match starting at $t[i]$ where $0 \leq i < n$, it starts by comparing letters $p[m-1]$ and

Algorithm 2.1 Naive string search algorithm

```
1: n ← length t
2: m ← length p
3: i ← 0
4: while i ≤ (n − m) do
5:     j ← 0
6:     while j < m and t[i + j] = p[j] do
7:         j ← j + 1
8:     if j = m then
9:         return i                          ▷ p is a factor of t
10:    i ← i + 1
11: return no valid result                   ▷ There is no factor p in t
```

$t[i+m-1]$. This allows us to move the word p to the right over more than just one letter when a mismatch is encountered. The shift value is precomputed by a number of rules. After each alignment, a rule which skips more letters is used.

2.2.1 Bad Character Shift

Consider an example:

```
t = I G U A N A S   E A T   B A N A N A S
        ≠ ‖ ‖ ‖
p = B A N A N A
```

Matching from the end of p to its start, a mismatch is encountered at $t[2]$ and $p[2]$. Since a letter "U" located at $t[2]$ does not appear anywhere in p, we can shift p by m and start looking for a match at $t[2+m]$:

```
t = I G U A N A S   E A T   B A N A N A S
    B A N A N A
          B A N A N A
                    B A N A N A
```

If a mismatch letter of t occurs in p, we need to shift p to the last occurrence of that letter in p:

```
t = I G U A N A S   E A T   B A N A N A S
    B A N A N A
          B A N A N A
                    B A N A N A
                      B A N A N A
```

The final alignments of the given pattern to the given text are:

```
t = I G U A N A S   E A T   B A N A N A S
    B A N A N A
          B A N A N A
                    B A N A N A
                      B A N A N A
                          B A N A N A
```

In order to get the number of positions to shift p to the right, the algorithm preprocesses p and creates a shift table.

The algorithm starts at the end of p with a count of 1 and it moves to the start of p. First time it moves left it does not change the count value. Every other time it moves left, it increases the count by 1. The current letter absent from the table is added, along with the current count. The letter that is already in the table is ignored. A unique letter that represents all other characters that does not appear in $alph(p)$ is added to the table along with a count m [13].

The shift table that stores the number of positions to shift the pattern *banana* is shown in Table 2.1.

A	**B**	**N**	*****
1	5	1	6

Table 2.1: Bad character shift table for a word `BANANA`

2.2.2 Good Suffix Rule

Let s denote the longest suffix of p that matches t in the current position. Good suffix rule defines 3 possible cases that are applied in the following order:

1. If there is a factor s' in p such that it is equal to s and it is not a suffix of p and the letter to the left of s' in p is not equal to the letter to the left of s in p, shift p to the right so that s' aligns with the factor s in t.

2. If there is no such factor, shift p to to the right to align the longest suffix of s in t with a matching prefix of p.

3. If none of above is possible, shift p by n places to the right.

2.2.3 Performance

When pattern occurs in the text, the worst case running time is $O(nm)$ and the best running time possible is $O(n/m)$. When pattern does not occur in the text, the algorithm requires $O(n + m)$ time.

2.3 Suffix Tree

A *tree* is a collection of *nodes* starting at a root node. Each node contains a value and a list of references to other nodes called *children*. No reference is duplicated and none points to the root node. A node with at least one child is an *inner node* and a node with no child is a *leaf*.

A *suffix trie* is a tree representing all suffixes of a word where every edge is labelled with a single letter. A suffix tree is a compressed suffix trie, where no two edges starting out at the same node can have the same prefixes. Meaning, individual edges may represent factors of a word longer than one letter. The difference of suffix trie and suffix tree is shown in Figure 2.1.

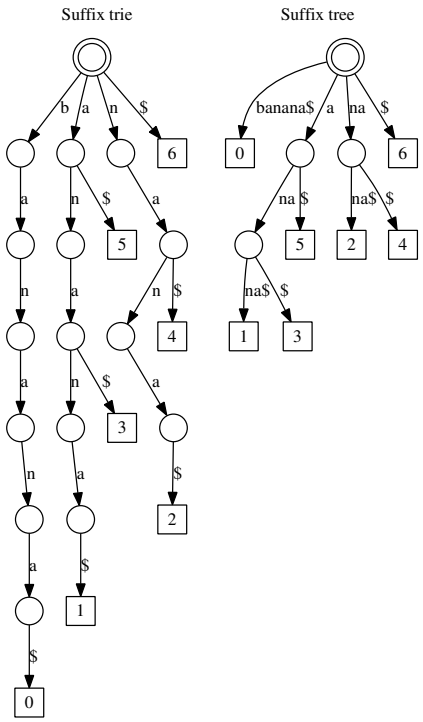

Figure 2.1: Difference between a suffix trie and a suffix tree for the word `banana`.

To avoid situations that a factor ends in an inner node and not in a leaf, a terminal letter that is unique within the word alphabet and is lexicographically smaller than all the other letters is added at the end of a factor. A suffix tree without a terminal letter is called an *implicit suffix tree*. Every leaf holds a number that represents the starting position of the corresponding suffix.

Let us count the number of nodes in a suffix tree for a word w of length n over a fixed alphabet Σ:

- There is exactly 1 root.
- Each suffix corresponds to a path from the root to a leaf so there are exactly $n + 1$ leaves with the terminal letter.
- Every inner node creates a new branch (with at least 2 children) that eventually leads to a leaf. There are $n + 1$ leaves in the suffix tree, therefore, the maximum number of inner nodes is n with the root. Since we do not consider the root as an inner node, the maximum number of inner nodes in the suffix tree is $n - 1$.
- Obviously, the maximum number of nodes in the suffix tree is $2n + 1$ which is the sum of the number of the root node, all leaves and all inner nodes of the suffix tree.

An example of a suffix tree with the maximum number of nodes is shown in Figure 2.2.

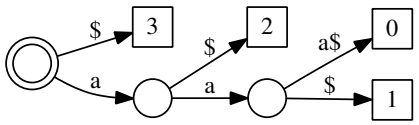

Figure 2.2: Suffix tree for the word `aaa`.

The construction of a suffix tree of w takes $O(n)$ using Ukkonen's algorithm. A reader who wishes to study this algorithm is advised to read [22].

Let us assume that $|\Sigma| \in O(1)$ and that one node also consumes $O(1)$ memory. The construction of a suffix tree of w takes $O(n)$ using Ukkonen's algorithm. The algorithm begins with an implicit suffix tree for the first character of w. A reader who wishes to study this algorithm is advised to read [22].

A suffix tree that has edge labels represented as factors of w will take $O(n^2)$ memory space. However, to avoid this situation we can use two numbers representing starting index and ending index of each factor of w. This decreases the memory space to $O(n)$. Thus, the overall memory space consumed by the suffix tree is $O(n)$ but this can vary from implementation.

2.3.1 Applications

Let us have a word p of length m.

To count all occurrences of p in w we follow the path for p starting from the root and we try to match p on a path. Three possible cases can occur:

1. The word p does not match.
2. The match ends in a node. All leaves below this node represent occurrences of p in w.
3. The match ends in an edge. All leaves below the ending node represent occurrences of p in w.

Finding p takes $O(m)$ time and collecting the leaves (for example by traversing a tree) takes $O(occ)$ time where occ is the number of occurrences of p in w. Thus, the overall time complexity of this algorithm is $O(m + occ)$.

To traverse a suffix tree, depth-first search algorithm can be used. Depth-first search algorithm visits the edges of a tree. We start in the root node. If we are visiting a node, then we next visit its children that has not yet been visited. If there is no such node, we return to the parent node. This is repeated until every node in a tree has been visited (Algorithm 2.2). Its worst case time complexity is proportional to the number of nodes in a graph. Thus, the worst case time complexity for a suffix tree of w is $O(n)$.

Algorithm 2.2 Depth-first search algorithm

```
1: procedure DFS(t, v)                          ▷ t is a subtree, v is a node
2:     v.discovered ← true
3:     m ← length P
4:     while all edges in t.edges(v) do
5:         if v.discovered = false then DFS(t, v)
```

Chapter 3

SageMath

SageMath is an open source mathematical software [18] that already supports combinatorics on words [20]. It contains a set of tools that we can build on and the opportunity to contribute to the user community, delivering a visible outcome of this thesis. These are the reasons why SageMath has been chosen as the software of choice for this thesis.

3.1 Introduction

SageMath is a free open source mathematical software created as an alternative to other proprietary mathematical software. Instead of implementing everything from scratch, SageMath integrates a number of established and narrow-specialized open source mathematical and statistical software [19] into a unified interface and it contains a lot of its own functionalities as well. It is intended to be easy to use and still to cover what is expected by any scientific tool, so it can be used both for studying and for research.

3.2 SageMath Development

SageMath development is driven by community of volunteers supported by various grants and mathematical associations.

It is written in several programming languages – *Python*, *Cython* and *C*. The source code is public and it is managed by *Git*.

SageMath is licensed under the GNU Public License [18] allowing commercial use. However, any modifications to the source code should stay under the same license agreement [9].

3.3 Words Relevant Classes

SageMath covers a broad spectrum of combinatorics [20]. In this section, we introduce classes which are connected with the notion of word and with our implementation. Python data containers list, tuple, set and dictionary are described in Appendix A.

3.3.1 Word Morphism

The class `WordMorphism` located in the `sage.combinat.words.morphism` module, represents a morphism as defined in Section 1.2. Internally, it is represented as a Python dictionary where every key-value pair is one rule of a given morphism.

Useful methods implemented in the `WordMorphism` class:

- *constructor*
  The constructor can handle string, list, dictionary or another `WordMorphism` object.

- `is_erasing()`
  Returns `True` if the given morphism is an erasing morphism, i.e. the image of a letter is the empty word. Otherwise returns `False`.

- `is_growing()`
  Returns `True` if a given morphism is a growing morphism. A morphism $\varphi = \mathcal{A}^* \mapsto \mathcal{A}^*$ is growing if

  $$\lim_{n\to\infty} |\varphi^n(w)| = +\infty, \quad \text{where } w \text{ is the given word.}$$

  Otherwise returns `False`.

The example below demonstrates these methods:

```
sage: m = WordMorphism("0->1,1->0"); m
WordMorphism: 0->1, 1->0
sage: m.is_erasing()
False
sage: m.is_growing()
False
```

3.3.2 Finite Word

Finite words are internally represented as `FiniteWord_class`. There are more than hundred methods defined for this class and its .

- `length()`
  Returns the length of a word.

- `is_empty()`
  Returns `True` if the word is ϵ, otherwise returns `False`.

- `reversal()`
  Returns the reversal word of the word.

- `apply_morphism(morph)`
  Applies the morphism `morph` to the word and returns the result.

  As described in Section 3.3.1, a `WordMorphism` object is internally implemented as a dictionary. It takes $O(1)$ time to get an item from a dictionary. Given a word of the length n and a morphism `morph`, the method applies all rules in the morphism to all n letters of the word. Therefore, the time complexity of this method is $O(n)$.

- `suffix_tree()`
  Creates an implicit suffix tree. Uses Ukkonen's algorithm for a linear-time suffix tree construction (See Section 2.3 for tree description).

- `factor_iterator(n)`
  Builds a suffix tree of a given word and uses a depth-first search algorithm to return all distinct factors of length `n`. If `n` is not set, all distinct factors are returned.

- `find(sub,start,end)`
  Returns the index of the first occurrence of `sub` in the factor of the word starting from the index `start` and ending at the index `end`. It uses Boyer-Moore algorithm (Described in Section 2.2). The worst case running time of this method is quadratic.

The example below demonstrates some of the methods available for `Word` instances:

```
sage: w = Word("0100100101"); w
word: 0100100101
sage: w.find("100",1)
1
sage: w.find("100",2)
4
sage: wm = w.apply_morphism(WordMorphism("0->1,1->0")); wm
word: 1011011010
sage: wm.reversal()
word: 0101101101
sage: t = Word("abba").suffix_tree()
sage: sorted(t.factor_iterator(2))
[word: ab, word: ba, word: bb]
```

Chapter 4

Problem Analysis

In this section we analyze possible implementations of G-defect calculation and discuss their efficiency.

Throughout this section, the following setup is considered: The input is a finite word w of length n over a fixed alphabet Σ and a set of morphisms and antimorphisms denoted by G. The number of morphisms is denoted by m and the number of antimorphisms is denoted by r. Since G is a finite group $m = r$.

4.1 Analysis

As shown in the Section 1.4, there are two equivalent characterizations of G-defect to be considered:

- **Method 1:** $D_G(w) = |w| + 1 - \#\mathrm{Pal}_G(w) - \gamma_G(w)$;
- **Method 2:** $D_G(w) =$ the number of G-lacunas in w.

Let us break down these characterizations into several subproblems.

4.1.1 Method 1

Considering Method 1, to obtain $\#\mathrm{Pal}_G(w)$, we have to loop through all factors of w, validate whether they are G-palindromes or not and count the number of unique classes of equivalence in the set of found G-palindromes. There is, at most, $n \times (n+1)/2$ non-empty factors in w. However, there is, at most, $n+1$ elements in $\mathrm{Pal}_G(w)$ since $D_G(w) \geq 0$.

To obtain $\gamma_G(w)$, we count the number of unique classes of equivalence in the set of one-letter factors that are not G-palindromes. There is exactly $alph(w)$ such factors and they generate at most $alph(w)$ unique classes of equivalence.

4.1.2 Method 2

Considering Method 2, we have to loop through every prefix of w and validate whether its last letter and its G-longest palindromic suffix are G-unioccurrent in the prefix or not. Let us be reminded that G-lps(w) is a suffix u of w such that u is a G-palindrome. To find G-longest palindromic suffix of a prefix, we have to check whether all suffixes of the prefix are G-palindromes or not. It is certain that iterating over all suffixes of all prefixes of w is iterating over all factors of w. For every prefix of w, there is exactly one G-longest palindromic suffix. Therefore, we validate G-unioccurrence of G-longest palindromic suffix of all prefixes exactly n times. From the definition of G-lacuna we also check G-unioccurrence of every letter of w, G-unioccurrence is checked $2n$ times in total.

4.1.3 Discussion

As discussed above, both methods loop through all factors of w to find G-palindromes. However, Method 1 later works with $O(n^2)$ elements and Method 2 with $O(n)$ elements. Counting the number of unique classes of equivalence (Method 1) can be simplified to the string comparisons. G-unioccurrence (Method 2) can be simplified to the string matching problem. Both these problems have the same asymptotic time complexity.

Let us focus on analysis and implementation of Method 2 since in the worst case, it is working with significantly less amount of data than Method 1.

4.2 Applying Morphisms and Antimorphisms

As described in Section 3.3.2, SageMath already implements an efficient method for applying a morphism to a given word with the time complexity $O(n)$. Antimorphism is a composition of the reversal mapping and a morphism. To obtain a word by applying an antimorphism, the same method is used as in case of applying a morphism. The obtained result is reversed. The asymptotic time complexity stays the same as in case of applying a morphism.

Since this action is very frequent and takes a significant time to run, it is suggested to precompute and store all words obtained through applying needed morphisms and antimorphisms.

4.3 Finding All G-palindromes in a Word

Finding all G-palindromes in a word means iterating over factors of the given word and decide whether they are G-palindromes or not. Let us recall that a factor u of w is G-palindrome if there exists an antimorphism $\Theta \in G$ such that $u = \Theta(u)$. Different solutions for the described problem follows.

4.3.1 Naive Solution

The most straightforward way is to iterate over all factors of w, apply all given antimorphisms to every factor and compare obtained words with the initial factor. The performance of this solution is:

- picking all possible starting and ending positions to loop over all factors of w takes $O(n^2)$ time;
- loop over all antimorphisms requires $O(r)$ time;
- applying an antimorphism to all letters of a factor of w takes $O(n)$ time;
- comparing two words takes $O(n)$ time.

Thus, the overall time complexity of this naive approach is $O(rn^4)$.

4.3.2 Suffix Tree

We can get all distinct factors of a word by traversing a suffix tree. It takes $O(n)$ time to construct a suffix tree using Ukkonen's algorithm and since a suffix tree has at most $2n+1$ nodes, traversal of this tree can be done in $O(n)$ time. However edges of a suffix tree can contain factors longer than one letter. Thus, we have to iterate over all letters of each edge to get all distinct factors of w.

The performance of individual steps of this algorithm is:

- construction of a suffix tree takes $O(n)$ time;
- getting all distinct factors from a suffix tree takes $O(n^2)$ time;
- loop over all antimorphisms requires $O(r)$ time;
- comparing two words takes $O(n)$ time.

The construction of a suffix tree is the pre computation step, so the overall time complexity is calculated as $O(n) + O(rn^3)$. Thus, this solution takes $O(rn^3)$ time and $O(n)$ memory space.

4.3.3 Dynamic Programming

If w contains a factor $u \in A^+$ which is G-palindrome for an antimorphism $\Theta \in G$, then a word tuv such that $t, v \in \mathcal{A}$ and $\Theta(t) = v$, $\Theta(v) = t$, is also G-palindrome. We can use this property to avoid re-computation in validating G-palindromes.

The idea is to create a 2D table for every antimorphism where rows represent length of a factor denoted by `len` and columns represent an index where a factor starts in a word denoted by `idx`. An entry at the index `[len-1][idx]`

stores a boolean value, whether a factor of the length `len` starting at the index `idx` in w is G-palindrome for a given antimorphism or not.

With three or more letter factor tuv, the precomputed value for u is stored at the index `[len-3][idx+1]`. If the value stored at this index is `False`, a current factor tuv is automatically not a G-palindrome. If the value is `True`, the antimorphism should be applied to the first and last letter of a factor and do just two comparisons.

Now, let us demonstrate the above idea by an illustrative example. Take a word $ABBAB$ and the antimorphism Θ determined by $A \mapsto B$ and $B \mapsto A$:

1. We iterate over all one and two letter factors of w, apply Θ and store whether an obtained word is equal to an initial word. The result can be seen in Figure 4.1.

A False	B False	B False	A False	B False
AB True	BB False	BA True	AB True	
ABB -	BBA -	BAB -		
ABBA -	BBAB -			
ABBAB -				

Figure 4.1: Demonstration of finding all G-palindromes in the word `ABBAB`, Step 1

2. For next three or more letter factors, stored values can be used. A factor ABB starts at the index 0 in w and its length is 3. Therefore, its value is stored at the index `[2][0]` in the table. As previously described, a value stored at the index `[0][1]` is used to decide whether the factor ABB can be G-palindrome or not. Since the value stored there is `False`, we can automatically say that the factor ABB is not G-palindrome for the given antimorphism (Figure 4.2).

3. Thus, if a precomputed value is `True` (as in case of a factor $BBAB$) and it is true that Θ(first letter) = last letter and Θ(last letter) = first letter, it is then concluded that a given factor is G-palindrome (Figure 4.3).

4. The final table of w for Θ is shown on Figure 4.4.

Let us discuss the performance of this solution:

- picking all possible starting and ending positions to loop over all factors of w takes $O(n^2)$ time;

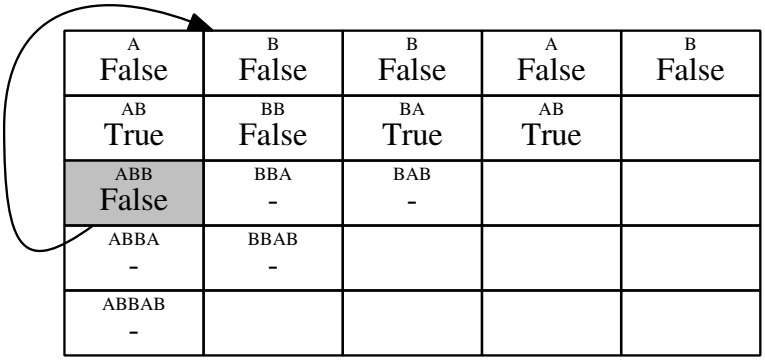

Figure 4.2: Demonstration of finding all G-palindromes in the word `ABBAB`, Step 2

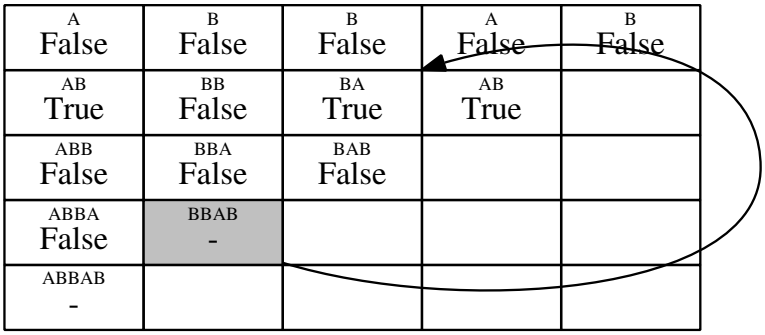

Figure 4.3: Demonstration of finding all G-palindromes in the word `ABBAB`, Step 3

- iterating over precomputed words obtained by applying all antimorphisms takes $O(r)$ time;
- applying an antimorphism to a letter takes $O(1)$ time;
- comparing two letters $O(1)$ time.

The proposed algorithm has time complexity $O(rn^2)$ and memory complexity $O(n^2)$.

Since we only need to store rows two steps back, we can change our table to store 2 rows and n columns. In that case a value whether a factor of the length `len` starting at the index `idx` in a given word can be G-palindrome is stored at the index `[len&1][idx]` in the proposed table, where `&` is a logical AND operator. Precomputed values for three and more letter factors are then stored at `[len&1][idx+1]`.

A False	B False	B False	A False	B False
AB True	BB False	BA True	AB True	
ABB False	BBA False	BAB False		
ABBA False	BBAB False			
ABBAB False				

Figure 4.4: Demonstration of finding all G-palindromes in the word `ABBAB`, Step 4

Thus, the overall time complexity of this solution is $O(rn^2)$ and the memory complexity is $O(n)$.

4.4 G-unioccurrence

Given a suffix of a word we want to find out whether it is G-unioccurrent in the word or not.

This can be seen as a string matching problem where the patterns are elements of classes of with finding all possible occurrences. We have introduced two possible algorithms in Chapter 2:

1. Boyer-Moore algorithm
2. Search in a suffix tree

Chapter 5

Implementation and Testing

5.1 Generating G

This method was implemented for easier generation of G for G-defect method.

- `generate_g(am)`: Generates subgroup of symmetries by list of antimorphisms `am`.
- **Input**
  - `am`: list of `WordMorphism` objects, growing or erasing morphisms are not accepted
- **Output**
  - Tuple of sets of morphisms and antimorphisms. As described in Section 1.3, G is a group but we do not need any group operations. Therefore, we represent G as a tuple.
- **Example**

```
m, am = generate_g([WordMorphism("0->0,1->1")])
```

5.2 G-defect

Let us be reminded, that we chose counting G-lacunas as the preferred characterization of G-defect. To implement this characterization, we have to loop through every prefix of a given word and validate whether its last letter and its G-longest palindromic suffix are G-unioccurrent in the prefix or not.

- `gdefect(w, g)`: Computes G-defect of the word `w` over `g`
- **Input**

  - `w`: word of length n represented as an instance of `Word` class
  - `g`: G represented as a tuple of sets of morphisms and antimorphisms. This can be easily generated from a list of antimorphisms by using `generate_g(am)`.

- **Output**
  - number representing the G-defect of `w`

- **Example**

```
m, am = generate_g([WordMorphism("0->0,1->1")])
```

5.2.1 G-longest Palindromic Suffix

Two promising methods to find all G-palindromes in w were analyzed in Section 4.3. The algorithm implemented with a suffix tree runs in $O(rn^3)$ time and the proposed dynamic programming solution runs in is $O(rn^2)$.

The output of this subproblem is a list of n numbers where each number at the index i represents the length of a G-longest palindromic suffix ending at the index i in w.

5.2.2 G-unioccurrence

Already implemented Boyer-Moore algorithm is used to search in a prefix of w (See detailed description of Boyer-Moore algorithm in Section 2.2).

- iterating over precomputed words obtained by applying all antimorphisms takes $O(r)$ time;
- the worst case scenario of Boyer-Moore algorithm takes $O(n^2)$ time;

Thus the overall time complexity of this subproblem is $O(rn^2)$. However we break the algorithm immediately after we find at least two occurrences of a pattern in w.

5.2.3 Overall Time Complexity

Time complexity of the subproblems is as follows:

- **G-longest palindromic suffix**: $O(rn^3)$;
- **G-unioccurrence**: $O(rn^2)$.

Thus, the final time complexity is $O(rn^3)$.

5.3 Testing

To load the implemented code, simply load a file `bp_guramatu.py` into your locally running SageMath application:

```
sage: load("<filepath">)
```

It was proven in [16], that Thue-Morse words has G-defect equal to 0. Thus, we automatically tested our implementation on huge number of Thue-Morse words as follows:

```
sage: for i in range(500):
....:     w = words.ThueMorseWord("01")[:i]
....:     if gdefect(w,g) != 0:
....:          raise Exception("Wrong result!")
```

All tests passed successfully. The implemented test method is called `gdefect_test()` and it is documented in the attached source code.

Conclusion

This thesis analyses and implements algorithms related to the recently introduced notion of palindromic richness with respect to a finite subgroup of symmetries generated by antimorphisms.

Combinatorics on words as a relatively new field of discrete mathematics and its main notions and definitions were presented.

We described and compared multiple algorithms and data structures for a wide variety of related string processing operations. We introduced the open source computational software SageMath and briefly presented its capabilities related to our topic.

We analyzed several algorithms and data structures for the purpose of designing an algorithm. Based on this analysis and on the available tools in SageMath, an algorithm for the computation of G-defect was designed and implemented. The presented solution is definitely better than naive and we are confident that we achieved a sufficient speed up. We believe that our contribution can help scientists achieve faster progress in their work.

Future Work

We want to include our work into SageMath, namely as a method of the class representing a finite word. Thus making the code available to all users of SageMath.

During our investigation we have found a multiple inefficient implementations of methods in SageMath and several bugs. We want to stay close to the SageMath community and contribute with the knowledge we gathered over the period of writing this thesis.

Bibliography

[1] TimeComplexity - Python Wiki [online]. `https://wiki.python.org/moin/TimeComplexity`, [Cited 2015-06-05].

[2] Allouche, J.-P.; Shallit, J.: Sums of Digits, Overlaps, and Palindromes. *Discrete Mathematics and Theoretical Computer Science*, volume 4, no. 1, 2000: pp. 1–10.

[3] Boyer, C. B.; Merzbach, U. C.: *A History of Mathematics*. Wiley, John & Sons, Incorporated, 1991, ISBN 0471543977.

[4] Boyer, R.; Moore, S.: A fast string searching algorithm. *Communications of the ACM*, volume 20, no. 10, 1977: pp. 762–772.

[5] Brlek, S.; Hamel, S.; Nivat, M.; etc.: On the Palindromic Complexity of Infinite Words. *International Journal of Foundations of Computer Science (IJFCS)*, volume 15, no. 2, 2004.

[6] Brlek, S.; Ladouceur, A.: A note on differentiable palindromes. *Theoret. Comput. Sci.*, volume 302, 2003: pp. 167–178.

[7] Charras, C.; Lecroq, T.: Exact string matching algorithms [online], [Cited 2015-06-05].

[8] Droubay, X.; Justin, J.; Pirillo, G.: Episturmian words and some constructions of de Luca and Rauzy. *Theoret. Comput. Sci.*, volume 255, 2001: pp. 539–553.

[9] Free Software Foundation: The GNU Public License v3.0 [online]. `http://www.gnu.org/licenses/gpl.html`, [Cited 2015-05-04].

[10] Glen, A.; Justin, J.; Widmer, S.; etc.: Palindromic richness. *European Journal of Combinatorics*, volume 30, 2009: pp. 510–531.

[11] Hof, A.; Knill, O.; Simon, B.: Singular continuous spectrum for palindromic Schrodinger operators. *Communications in Mathematical Physics*, volume 174, no. 1, 1995: pp. 149–159.

[12] Kari, L.; Mahalingam, K.: Watson-Crick palindromes in DNA computing. *Natural Computing*, , no. 9, 2010: pp. 297–316.

[13] Korber, E.: String Matching Algorithms. `http://www-inst.eecs.berkeley.edu/~cs61b/su06/lecnotes/lec28.pdf`, August 2006.

[14] Lothaire, M.: *Combinatorics on Words*, *Encyclopedia of Mathematics and its Applications*, volume 17. Addison-Wesley, Reading, Mass., 1983, reprinted in the *Cambridge Mathematical Library*, Cambridge University Press, UK, 1997.

[15] Manacher, G.: A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *J. ACM*, volume 22, no. 3, 1975: pp. 346–351.

[16] Pelantová, E.; Starosta, S.: Languages invariant under more symmetries: overlapping factors versus palindromic richness. *Discrete Math.*, volume 313, 2013: pp. 2432–2445.

[17] Pelantová, E.; Starosta, S.: Palindromic richness for languages invariant under more symmetries. *Theoret. Comput. Sci.*, volume 518, 2014: pp. 42–63.

[18] Stein, W.; etc.: *Sage Mathematics Software [online]*. The Sage Development Team, [Cited 2015-05-05].

[19] Stein, W.; etc.: SageMath - Components [online]. `http://www.sagemath.org/links-components.html`, [Cited 2015-05-04].

[20] Stein, W.; etc.: Words – Sage Reference Manual [online]. `http://www.sagemath.org/doc/reference/combinat/sage/combinat/words/__init__.html`, [Cited 2015-05-05].

[21] Thue, A.: Über unendliche Zeichenreihen. *Norske Vid. Selsk. Skr. I Math-Nat. Kl.*, volume 7, 1906: pp. 1–22.

[22] Ukkonen, E.: On-line construction of suffix trees. *Algorithmica*, volume 14, 1995: pp. 249–260.

Appendix A

Python Data Containers

Python provides several general purpose built-in data containers. We introduce them and we discuss their time complexity classes to better decide in the future whether they suit our needs or not. In all tables below, n is the total number of items in a described data container. Time complexity of these containers were described in [1].

A.1 List

A data type `list` holds a list of items which do not need to be the same type in a given order. These items are separated by commas and enclosed within square brackets. A list is indexed from zero to the total number of items minus one.

Index	**Example**	**Time complexity**
Get Item	`l[i]`	$O(1)$
Set Item	`l[i] = 1`	$O(1)$
Append	`l.append(1)`	$O(1)$
Length	`len(l)`	$O(1)$
Containment	`x in l`	$O(n)$
Sort	`l.sort()`	$O(n \log n)$

Table A.1: Time complexities of selected list operations

A.2 Tuple

A data type `tuple` can be expressed as a read-only list since items within cannot be updated. Therefore, time complexities for all their operations are the same as their list equivalents (Table A.1). Items are enclosed in parentheses.

A.3 Set

A data type `set` stores only unique unordered elements.

Index	**Example**	**Average Case**	**Amortized Worst Case**
Add Item	`s.add(0)`	$O(1)$	$O(n)$
Containment	`x in s`	$O(1)$	$O(n)$

Table A.2: Time complexities of selected set operations

A.4 Dictionary

A data type `dict` is an unordered set of key-value pairs with unique keys. It is implemented using hash tables therefore, keys have to be hashable. A dictionary is enclosed in curly braces and its values can be accessed using square braces.

Index	**Example**	**Average Case**	**Amortized Worst Case**
Get Item	`d[k]`	$O(1)$	$O(n)$
Store Item	`d[k] = v`	$O(1)$	$O(n)$
Delete Item	`del d[k]`	$O(1)$	$O(n)$

Table A.3: Time complexities of selected dictionary operations

Appendix B

Contents of CD

The visualised content of the enclosed media.

- `src` ..... the directory of source codes
  - `figures` ..... the thesis figures directory
  - `*.bib` ..... the bibliography source code files of the thesis
  - `*.tex` ..... the LaTeX source code files of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the Diploma thesis in PDF format
- `bp_guramatu.py` ..... the source code of the implemented solution